**Top of Form**

**The problem with disabling buttons is that... it removes them from the tab index.**

# Principles of Accessibility

Accessibility principles are guidelines and best practices aimed at ensuring that products, information, and environments are usable regardless of users’ abilities, backgrounds, or circumstances. Here are the leading principles:

**1. Perceivable**

* Definition: Information and components must be presentable to users in ways they can perceive.
* Examples:
  + Providing text alternatives for images (alt text).
  + Using sufficient color contrast for readability.
  + Captioning videos for those who are deaf or hard of hearing.
* Why it matters: Ensures everyone can access content, regardless of their sensory abilities.

**2. Operable**

* Definition: Navigation and interface components must be usable by all, including those relying on keyboards or assistive devices.
* Examples:
  + Ensuring all functions are accessible via keyboard, not just mouse.
  + Giving users enough time to read and use content.
  + Avoiding features that could trigger seizures (like flashing content).
* Why it matters: Reduces barriers for people with motor impairments or cognitive limitations.

**3. Understandable**

* Definition: Content and operation must be comprehensible and predictable.
* Examples:
  + Simple, clear language.
  + Consistent navigation and layout.
  + Providing helpful error messages and guidance.
* Why it matters: Makes experiences inclusive for those with cognitive or learning disabilities, as well as non-native speakers.

**4. Robust**

* Definition: Content must be compatible with current and future technologies, including assistive devices.
* Examples:
  + Using clean, valid code so screen readers can interpret the content accurately.
  + Supporting a variety of browsers and devices.
* Why it matters: Ensures accessibility remains effective as technologies evolve.

# Types of Images

What types of images are commonly used in web development and accessibility?

1. **Informative Images:**  
   Convey essential content or information (e.g., charts, infographics), must have meaningful alt text describing their purpose.
2. **Decorative Images:**  
   Used for aesthetic purposes only (e.g., background graphics), should have empty alt attributes (alt="") so they’re ignored by screen readers.
3. **Functional Images:**  
   Used as interactive controls (e.g., buttons, icons), alt text must reflect their action (e.g., “Search”).
4. **Complex Images:**Diagrams, graphs, or images that need more descriptive text; may require additional explanation outside the alt attribute, such as a long description or caption.

**Bottom of Form**

**1. What is web accessibility and why is it important?**

Answer:  
Web accessibility means designing and developing websites and applications so people with disabilities can use them. It ensures equal access to information and functionality for everyone, regardless of ability. It’s essential for inclusion, is often legally required, and improves overall usability for everyone (e.g., via clear navigation and readable content).

**What does semantic tag implementation mean?**

Semantic tags in HTML describe the meaning and structure of content, making it easier for browsers, assistive technologies (like screen readers), and developers to understand and render web pages. Implementing semantic tags means using the correct HTML elements for content (e.g., <header>, <nav>, <main>, <article>, <section>, <footer>, <aside>), rather than generic tags like <div> or <span>.

**4. What is ARIA? When should ARIA attributes be used?**

**Answer:**  
**ARIA (Accessible Rich Internet Applications)** attributes add extra information about UI elements for assistive technologies. Use ARIA when semantic HTML alone cannot provide enough meaning (like for custom controls or dynamic content). Prefer native HTML elements over ARIA when possible.

**Benefits of Semantic Tags**

Why are semantic tags beneficial?

* Accessibility:  
  Help assistive technologies (screen readers) determine the type and hierarchy of content, improving navigation for users with disabilities.
* SEO:  
  Search engines use semantic tags to understand and rank content accurately. Semantic HTML helps improve SEO by making the content of websites more identifiable to search engines.
* Maintainability:  
  Provide clear structure to code, making it easier to read, update, and maintain.
* Interoperability:  
  Better compatibility with future devices, platforms, and technologies.
* Readability:

Developers often work on teams, so it’s important their code is readable — and semantic HTML helps structure and organize your code to keep everyone on the same page.

**What are some tools do you use for accessibility testing?**

**Answer:**  
I use automated tools (axe, WAVE, Lighthouse), screen readers (JAWS, NVDA, VoiceOver), Color Contrast Analyser, and perform keyboard-only navigation to check for focus order and operability. Manual testing complements automated checks for a thorough evaluation.

**How would you handle dynamic content updates for screen readers?**

**Answer:**  
Use WAI-ARIA live regions (aria-live, aria-atomic, aria-relevant) to notify screen readers about content changes. Choose the appropriate value: polite for non-critical updates, assertive for urgent alerts. Test that updates are communicated without disrupting user flow. Prefer native HTML solutions (e.g., form validation using <output>) where possible.

**Explain how accessibility can impact SEO.**

**Answer:**  
Semantic HTML and proper labeling (headings, alt text, landmarks) benefit both screen readers and search engines—making content discoverable and properly indexed. Accessible websites often have faster load times and improved structure, both of which boost SEO.

**What are the challenges with ARIA, and how do you mitigate them?**

**Answer**:  
ARIA is powerful but easy to misuse—incorrect roles or attributes can break accessibility. Overuse is risky; always use native HTML elements first. When ARIA is required, test with multiple assistive technologies, maintain documentation, and validate code using tools like axe and the W3C ARIA validator.  
  
ARIA (Accessible Rich Internet Applications) attributes

**ARIA (Accessible Rich Internet Applications)** attributes are HTML features that help make web content and apps more usable for people with disabilities—especially those using screen readers.

They describe roles, states, and properties of UI elements, enabling enhanced accessibility, especially for dynamic content and custom widgets.

ARIA Attribute Reference & Examples

Below is a list of core ARIA attributes as found in the MDN ARIA Reference (Note: I cannot provide direct links, but this list is comprehensive and tailored for learning and production use.)

**1. aria-label**

Description: Provides a text label for an element, helping assistive technology explain its purpose.

<button aria-label="Close dialog">✖</button>

**2. aria-labelledby**

Description: References another element that serves as the label. Use when another on-page element is the label.

<h2 id="menuTitle">Menu</h2>

<nav aria-labelledby="menuTitle">

...

</nav>

**3. aria-describedby**

Description: References another element that adds extra descriptive information.

<input type="text" aria-describedby="desc">

<div id="desc">Please enter your full name.</div>

**4. aria-hidden**

Description: Hides the element from assistive technologies.

<div aria-hidden="true">

<!-- Decorative icon, not important for screen readers -->

<img src="icon.png" alt="">

</div>

**5. aria-disabled**

Description: Indicates that the element is disabled (not interactive).

<button aria-disabled="true">Submit</button>

NOTE: The problem with disabling buttons is that... it removes them from the tab index.

**6. aria-checked**

Description: Indicates the checked state of checkboxes, radio buttons, or switches.

<input type="checkbox" aria-checked="false">

**7. aria-selected**

Description: Marks an item as selected inside grouped (e.g., tab, listbox) controls.

<li aria-selected="true">Home</li>

<li aria-selected="false">Profile</li>

**8. aria-expanded**

Description: Indicates whether a collapsible element is open ("expanded") or closed ("collapsed").

<button aria-expanded="false" aria-controls="section1">More Info</button>

<div id="section1" hidden>

...

</div>

**9. aria-controls**

Description: Identifies the elements whose visibility or state the current element controls.

<button aria-controls="menuPanel">Toggle Menu</button>

<div id="menuPanel">...</div>

**10. aria-invalid**

Description: The aria-invalid attribute is used to indicate that the value entered into an input field is not in a format or a value the application will accept.

<input type="number" step="2" min="0" max="100" aria-required="true" aria-invalid="false" />

**11. aria-busy**

Description: Indicates the element (or its children) are being updated and should not be read yet.

<div aria-busy="true">Loading data...</div>

**12. aria-valuenow, aria-valuemin, aria-valuemax**

Description: Provide current, min, and max value for range widgets (like sliders).

<div role="slider" aria-valuenow="25" aria-valuemin="0" aria-valuemax="100"></div>

**13. aria-required**

Description: Indicates an input is required.

<input aria-required="true">

**14. aria-role**

Description: Not an attribute but used via the role attribute to tell assistive tech what type of widget this is (e.g., "button", "navigation", "progressbar").

<div role="progressbar" aria-valuenow="30" aria-valuemin="0" aria-valuemax="100"></div>

Role is typically set with the role attribute and is essential for ARIA to work.

**15. aria-pressed**

Description: Indicates the pressed state of toggle buttons.

<button aria-pressed="true">Bold</button>

**16. aria-modal**

Description: Indicates whether a dialog is modal.

<div role="dialog" aria-modal="true">

...

</div>

**17. aria-activedescendant**

Description: Identifies which child element of a composite widget is currently active.

<ul role="listbox" aria-activedescendant="item2">

<li id="item1">Item 1</li>

<li id="item2">Item 2 (Active)</li>

</ul>

**18. aria-autocomplete**

Description: Provides auto-completion guidance for textboxes.

<input type="text" aria-autocomplete="list">

Values: "none", "inline", "list", "both"

**19. aria-haspopup**

Description: Indicates the presence of a sub-level popup (menu, listbox, tree, grid, dialog).

<button aria-haspopup="menu">Options</button>

Values: "false", "true", "menu", "listbox", "tree", "grid", "dialog"

**20. aria-orientation**

Description: Indicates the orientation of a widget (horizontal/vertical).

<div role="slider" aria-orientation="vertical"></div>

#### ARIA Live Region Attributes:

When websites or apps dynamically update content (e.g., notifications, live results), those changes aren’t always detected by screen readers. ARIA live region attributes solve this problem by letting developers declare which content updates should be announced, and how.

**21. aria-live**

Defines how assertively changes to an element should be announced to assistive technologies (like screen readers).

Values:

off (default): Updates are not announced.

polite: Updates are announced when the user is idle, without interrupting their current activity.

assertive: Updates are announced immediately, potentially interrupting what the screen reader is currently saying.

Example:

HTML, XML<div aria-live="polite">

<!-- Updates here will be announced politely -->

</div>

When to use:

Polite: For non-critical updates (e.g., shopping cart changes, search results).

Assertive: For urgent alerts (e.g., error messages, warnings).

**22. aria-atomic**

Controls whether only the changed part of the element or the entire element is announced when it updates.

Values:

false (default): Only the changed portion is announced.

true: The entire region is re-announced, even if only a small part changes.

Example:

HTML, XML<div aria-live="polite" aria-atomic="true">

Items in your cart: 2

</div>

When to use:

Use when piecemeal updates might confuse users (e.g., totals or statuses where complete context is preferred).

**23. aria-relevant**

Specifies what kinds of changes (additions, removals, text changes) should be announced in the live region.

Values:

additions — new child elements added.

removals — child elements removed.

text — text content changes.

Can be comma-separated for multiple types (e.g., additions text).

Example:

HTML, XML<div aria-live="polite" aria-relevant="additions text">

<!-- Both new elements and text changes will be announced -->

</div>

When to use:

Tailor for your region’s needs—for instance, if items are added, removed, or edited in real-time (chat windows, notifications).

# BEST PRACTICES:

**Use semantic HTML:**

Structure content with elements like <h1>, <nav>, <main>, <button>, and <form>—not just <div> or <span>.

**Ensure keyboard accessibility:**

All interactive elements must be reachable and usable via keyboard (Tab, Enter, Space, Arrow keys).

**Provide visible focus indicators:**

Style: focus with outlines or highlights so users always know which item is active.

**Text alternatives for images:**

Always add descriptive alt text for images and icons.

**Accessible forms:**

Label every form field; use <label for="id">, clear error messages, and logical tab order.

**Color contrast:**

Text and important UI elements must have strong contrast with their backgrounds.

AA

4.5:1 contrast for normal text

3:1 for large text (>=24px)

AAA

7:1 contrast for normal text

4.5:1 for large text (>=24px)

**Do not rely on color or sound alone:**

Use more than just color to convey meaning (add labels, icons, patterns).

**Accessible media:**

Provide captions and transcripts for videos and audio. Include audio descriptions for key visuals if possible.

**ARIA only as needed:**

Use ARIA roles and properties to enhance—never replace—semantic HTML.

**Responsive and scalable layouts:**

Make sure content works with zoom (up to 200%) and on all screen sizes.

**Skip navigation links:**

Let keyboard users bypass repetitive page elements (<a href="#main">Skip to main content</a>).

**Logical structure and heading hierarchy:**

Use headings in the correct order for easy navigation (<h1>, <h2>, etc.).

**Test with screen readers and real users:**

Use tools like NVDA, JAWS, VoiceOver, and get diverse feedback.

**Avoid moving or auto-updating content without control:**

Provide pause, stop, or controls for animation, carousels, and notifications.

**Time limits:**

Give users ways to extend or turn off timers in forms or applications. **Assistive Technologies:**

Screen Readers

Speech recognition software

Screen magnifiers

Alternative input devices

**Accessibility Testing:**

Contrast Checkers

Automated Tools:

Lighthouse

**Color Contrast:**

WCAG: A level, AA level and AAA level

AA

4.5:1 contrast for normal text

3:1 for large text (>=24px)

AAA

7:1 contrast for normal text

4.5:1 for large text (>=24px)

# Accessibility Testing Tools

Absolutely! Here’s a comprehensive overview of **accessibility testing tools**, including what they are, popular examples, and practical considerations for selecting the right ones. This will prepare you for interview questions about how accessibility is tested in modern web and application development.

**What Are Accessibility Testing Tools?**

Accessibility testing tools help teams evaluate websites, apps, and digital content for compliance with accessibility standards (like WCAG), and for usability by people with disabilities. These tools can highlight issues, automate checks, and support manual testing—helping developers, designers, and testers create inclusive digital experiences.

**Categories of Accessibility Testing Tools**

**1. Automated Evaluation Tools**

These scan pages for accessibility errors based on standards (e.g., missing alt text, contrast issues).

* **Examples:**
  + **axe** (Browser extension/CLI): Popular, integrates with development workflows.
  + **WAVE** (WebAIM): Browser extension providing visual feedback.
  + **Lighthouse** (Google Chrome DevTools): Automated audit includes accessibility checks.
  + **Accessibility Insights** (Microsoft): Browser plugin offering fast automated and guided tests.
  + **pa11y**: CLI and dashboard for automated accessibility testing.

**2. Screen Readers**

Used for manual, “real world” testing to ensure users reliant on assistive technology can navigate and understand content.

* **Examples:**
  + **JAWS:** Professional-grade for Windows.
  + **NVDA:** Free, open source for Windows.
  + **VoiceOver:** Built into macOS and iOS.
  + **TalkBack:** Android devices.
  + **Narrator:** Windows built-in option.

**3. Color Contrast Analyzers**

Check for readable color combinations.

* **Examples:**
  + **Colour Contrast Analyser (TPGi):** Desktop app to test color contrast ratios.
  + **axe and WAVE:** Also check for contrast issues.

**4. Keyboard Navigation Testing**

Not a specific tool, but a manual practice:

* Ensure all interactive elements (links, buttons, forms) are accessible via keyboard without a mouse.

**5. Code Linters & Static Analysis**

Integrated into CI/CD or editor workflows to catch issues early.

* **Examples:**
  + **eslint-plugin-jsx-a11y:** For React projects.
  + **Accessibility add-ons for Storybook:** Check UI component libraries.

**Practical Considerations**

* **No single tool finds all issues:** Combine automated tools with manual testing (especially with screen readers and keyboard navigation).
* **Regular testing:** Accessibility should be checked in every sprint—not just at the end.
* **Human judgment matters:** Automated tools can miss context-specific issues or usability concerns.

**Typical Interview Talking Point**

*"To ensure accessibility, we use automated tools like axe for regular scans, and then validate with manual tests using screen readers (NVDA) and keyboard-only navigation. For color contrast, I rely on Colour Contrast Analyser. Automated tools catch obvious failures, while manual testing ensures usability for real users."***How do you test keyboard accessibility?**

**Answer:**  
Navigate the site using only the keyboard (Tab, Shift+Tab, Enter, Space, arrow keys). Ensure that all interactive elements are reachable and usable, and that focus order is logical. Visible focus indicators should show where the user is on the page.

**Keyboard Accessibility: Key Functions and Their Roles**

When testing a website or app for keyboard accessibility, you use a set of standard keys to simulate how users without a mouse navigate and interact. Here’s what each key does and what to check for:

**Tab**

* **Purpose:** Moves keyboard focus forward to the next interactive element (links, buttons, form fields, etc.).
* **Testing:**
  + Pressing Tab should sequentially move through all interactive elements in a logical order (e.g., top to bottom, left to right).
  + No elements should be skipped.
  + The page should show a visible indicator (like an outline) on the currently focused element.
* **Why it matters:** Ensures users can reach all controls without a mouse and know where they are on the page.

**Shift + Tab**

* **Purpose:** Moves keyboard focus backward to the previous interactive element.
* **Testing:**
  + Use Shift+Tab to reverse through the focusable elements.
  + Check that backward navigation is seamless and logical, matching the Tab order.
* **Why it matters:** Allows users to return to previous items without starting over, essential for forms or navigation menus.

**Enter**

* **Purpose:** Activates the focused element, such as submitting a form, following a link, or triggering a button.
* **Testing:**
  + Tab to a button, link, or interactive control and press Enter—ensure it performs its expected action.
* **Why it matters:** Users must be able to interact fully using only the keyboard.

**Space (Spacebar)**

* **Purpose:**
  + Primarily toggles checkboxes, radio buttons, or activates certain controls (like buttons).
  + In some UI components (e.g., expandable panels), Spacebar opens/closes items.
* **Testing:**
  + Tab to checkboxes/radios and press Space—should toggle their state.
  + For custom widgets (accordions, menus), Space may open/close or select options.
* **Why it matters:** Ensures all controls are operable and state changes are accessible.

**Arrow Keys (Up/Down/Left/Right)**

* **Purpose:** Used for navigating inside certain components like menus, dropdown lists, sliders, or tab panels.
* **Testing:**
  + For components like dropdowns, sliders, tab lists, carousels, or menus—use arrow keys to move between, select, or activate choices.
  + Arrow key navigation should be intuitive and follow common patterns.
* **Why it matters:** Custom widgets or navigation areas must behave predictably for keyboard users, supporting deeper interaction within a component.